The Report Committee for Nithin Krishna Nanchari
Certifies that this is the approved version of the following report:

Austin Logistics Inc – Assessing Defect Density

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Dewayne Perry

Herbert Krasner

Austin Logistics Inc – Assessing Defect Density

by

Nithin Krishna Nanchari, B.Tech

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2010

Dedication

This thesis is dedicated to my father and mother who taught me that the best kind of knowledge to have is that which is learned from its own sake.

Also, this thesis is dedicated to my wife who has been a great source of motivation and inspiration. I attribute the level of my Masters degree for her patience, encouragement and support all throughout my course work.

Acknowledgements

I am heartily thankful to Dr. Herbert Krasner, whose encouragement, guidance and support from the initial to the final enabled me to develop an understanding of the subject.

Lastly, I offer my regards to all those who supported me in any respect during the completion of the project.

December 2010

Abstract

Austin Logistics Inc – Assessing Defect Density

Nithin Krishna Nanchari, M.S.E

The University of Texas at Austin, 2010

Supervisor: Dewayne Perry

Austin Logistics Inc. Solutions provides tools that help centralize resource management, optimize and maintain compliance of calling schedules for consumer financial service organization(Banks, financial institutions). With the increasing number of customers, the amount of rework and availability of resources had been notably decreasing over time; thereby negatively affecting the overall cost and quality of the software being delivered. The improvement objectives of the company and its departments were broadly stated but lacking a goal-driven nature. The software measurement Goal-Question-Metric (GQM) approach was chosen and used for this research initiative to better support business driven quality improvement. Software defect density data was collected and analyzed to identify significant deviations in the software development life cycle.. The results of the initial analysis on the transformed defect-tracking data helped identify the negatively affected areas within the software development life cycle. The data showed significant variations in the requirements, design and implementation phases of the product life cycle, thus helping identify various process improvement opportunities. On quantifying the change in defect density, the effectiveness of using GQM has also provided valuable insights for process improvement. Based on these results, we were able to identify some of the weaknesses and shortcomings in our application development process.

Table of Contents

List of Figures

Context

Austin Logistics Inc. Solutions (ALI. Inc Solutions) provide collection analytic applications that empower consumer financial service organizations (banks and other financial collection agencies) to help maximize their resource productivity by upholding the compliance policies laid out by the US government. Every application developed has different components and caters to the varied needs of different collection agencies. The applications built with analytic reasoning models help to identify not only the appropriate time a person could possibly be contacted (based upon the previous contact history), but also adjust dynamically for changes in both company resource requirements and policies. Some of the applications are developed to help automated call centers make appropriate resource management decisions while adjusting dynamically to changes in needs and policies; thus improving the effectiveness of customer outreach and optimizing resource usage rates.

The applications developed by ALI. Inc Solutions are broadly categorized into two product lines: Intelligent Contact Suite and Action Optimizer Suite. OnQ is one of the applications from the Intelligent Contact Suite that serves collection-oriented organizations (banks, call centers etc) and manages automated dialers. It can adjust dynamically to changes in resource requirements while continuing to uphold all the rules and compliance policies defined by the organization and the US government. OnQ has been in the marketplace for approximately 10 years.

Currently ALI. Inc Solutions is working on OnQ version 3.2.2. It is predominantly a JAVA oriented application. Recently, our company implemented a software tool that helps predict better release schedules, thereby improving and enabling more frequent delivery of the application. To determine the degree of portability of the tool, it was tested using various operating systems, databases, etc.

As the number of OnQ customers increased, new feature requests increased over time. Many of these requests were addressed during the normal application release cycle. Due to this change in the application requirements, issues relating to portability, performance and scheduling started growing with time. These issues caused persistent resource availability problems affecting the predictability of application delivery schedules.

The following software process tools were being used to help predict the application release schedule:

1. JIRA and Anthill to build and evaluate software release process
2. Clover to evaluate testing release code coverage only for Unit Testing
3. TestLink to view the test cases relating to functionality
4. Microsoft Project to evaluate application release schedules

These tools did not collect, calculate nor offer quantitative information used to compute software metrics that could assist the organization when making appropriate release decisions. They were also neither satisfactory nor sufficient in evaluating the product reliability, efficiency, and probable failure rate. Hence they were not useful in helping to predict the appropriate software release schedules.

Collecting code, reliability and test metrics to evaluate the application integrity and quality, was hypothesized as the way to help the organization make justifiable and rightful decisions about OnQ including release schedules and process improvement procedures.

Introduction to the Goal-Question-Metric (GQM) approach

Software engineering has been defined as "the disciplined application of engineering, scientific, and mathematical principles, methods, and tools to the production of quality software" [Humphrey 89][1]. Various phases of software engineering include planning, designing, implementing, testing, managing and maintenance. The business objectives of the above phases are often broad but lack goal-driven approach. Thus to improve each phase of the software life cycle, measurement process is necessary.

It has been shown that the prospects for company success with software development improve when decisions can be made based on factual and quantitative information obtained by the software measurements.

The GQM approach was developed in response to the need for a goal-oriented approach that would support the measurement of processes and products in the software engineering life cycle. The complexity of software engineering is increasing with time. One of the challenges of software engineering is to measure individual facets of a complex application, simultaneously, without losing focus on the primary objective. The GQM approach supports a top down approach to defining the goals behind measuring processes and products, and using these goals to decide precisely what to measure, namely choosing specific and well-defined metrics.

The GQM process emphasizes the need to

- Establish an explicit measurement goal specific to the process to be assessed
- Define a set of questions that must be answered to achieve the goal
- Identify formulated metrics that help answer the questions

Fig I below represent the basic GQM model defined by Basili

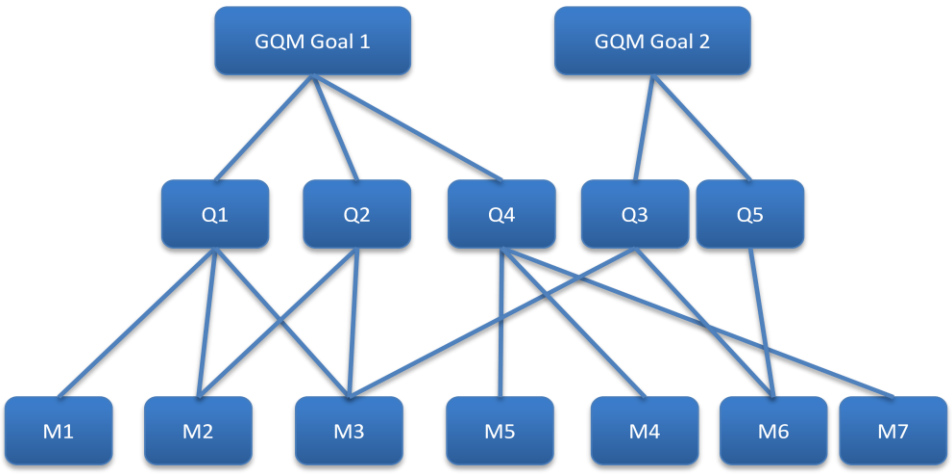

Fig I: The GQM Paradigm

To apply GQM to measure software development with respect to business goals, the following steps are needed [1]

Step I: Identify the business goal and the associated measurement goal from the company viewpoint.

Step II: Generate meaningful questions from mental models that define the goal as completely as possible in quantifiable way.

Step III: Group and translate the questions into relative sub-goals.

Step IV: Refine the mental model and the entities and associated attributes.

Step V: Translate issues and goals into measurement goals.

Step VI: Quantify the questions and formulate indicators that support the questions.

Step VII: Associate entities and attributes to formulated indicators.

Step VIII: Define need, usage and boundaries of both entities and attributes.

Step XI: Translate the necessary data elements into an action plan.

Step X: Define the process and prepare the plan to achieve the business goal

In order to cope with the issues related to the quality of the software described in the context section, we introduce the GQM approach as the measurement framework for this research project.

Step I: Identify the business goal

The goal-driven-metric approach (GQM) begins with identifying the business goal that initiates this measurements process. “Business goals are often a function of where you sit” [Lynch 91]. Setting a business goal for the GQM process provides a detailed guideline to what we intend to achieve. The following goal was identified after discussion with the team members

Goal: To identify the reasons for reduced project efficiency, resource productivity and availability during the software life cycle.

Step II: Identify what we want to learn

To understand the activities related to achieving the business goal, resources and processes affected were identified. Based on the mental models of the processes and the resources affected, questions are gathered and then grouped into different entities. Upon isolating each entity activities for the below collected questions, flow paths of the processes were then determined. A further assessment based on the products and by products provided more information about the processes that will help fulfill the company's goal. By identifying these questions and their grouping we will be able to understand, assess, predict or improve the activities relating to achieving the business goal.

Persons affected are:

a) Executives – Engineering Manager, Product Managers/Specialists
b) Project Managers – Track project, Estimate product release schedule
c) Developers – Implement required features
d) QA – Test the product, Open issues

Processes affected are:

a) Requirement: Scope and define project boundaries. Specify the functionalities that the product should adhere to.
b) Architecture and System Design: Specify how the software should perform for supported platforms.
c) Unit/Integration Testing: Testing isolated and interaction between modules.
d) System Testing: Scalability, Stress testing including performance evaluation.
e) Maintenance: Fixing failures. Software evolution as changing requirements.

Inputs and Resources

a) Hardware
    i. Is the hardware compatible?
    ii. Is it configurable?
    iii. What is the maintenance cycle requirement?
b) Software
    i. Use of an IDE and its effectiveness?
    ii. What is the frequency of Third party tool usage?
    iii. Do we reuse code from past projects?
c) Resource
    i. Do we have a competent development team?
    ii. Do they like their jobs?
    iii. Are they motivated/challenged?
    iv. Is the work environment ideal?
    v. Are they equipped with proper tools to perform their jobs?
    vi. Are they working across projects/clients?
d) Project Requirements
    i. Is there a requirement gathering process established?
    ii. What is the frequency of requirement change?
    iii. Are they changing closer to deadlines?
    iv. How comprehensible are the requirements?
    v. Is there a correlation between number of bugs and changing requirements?

Internal Artifacts

a) Software
    i. Is the development environment up to date?
    ii. Is the development and testing environments integrated?
b) Resource

i. Do we have past experience in the project domain?
ii. Do we have a documentation team?
iii. Do we have dedicated business analysts?
iv. How is the interaction between development and QA team?

c) Requirements

i. Do we have a process to acquire commitments on requirements?
ii. Does the team adhere to the project schedule?
iii. Do we have control over the requirements?
iv. How complex are new requirements?
v. When do we say no to new requests? Do we have a concept of "requirements freeze"?
vi. Do defects affect future requirements?

d) Measurements and metrics

i. Do we have a process to collect metrics during project development?
ii. Do we have existing metrics from past projects?
iii. Are any metrics applicable for present and future projects?
iv. Do we comprehend our software measurements?

Activities and Flow paths

a) Requirement

i. Are the development, architecture and design team involved in requirements gathering?
ii. How often are requirements triaged?

b) Estimation/ Budget Scoping

i. Are we utilizing past metrics for resource cost analysis?
ii. What techniques/models are used in estimation?
iii. Is empirical data being considered for estimation?
iv. Can you support your estimates with real data?
v. Are there any deadlines?

c) Product design

i. Is product design communicated within development/engineering team?
ii. Is our design comprehensible?
iii. Does it satisfy architectural requirements?
iv. Is enough time allocated for design analysis?
v. How is design evaluated?
vi. How does defects affect design?

d) Development

i. Are the design documents logical?
ii. Are code standards enforced?
iii. Do we perform code reviews?
iv. Do we perform proof of concept for a feature?
v. How efficiently do we use source control?
vi. Are developer changes tracked?
vii. Are we equipped with efficient build system?
viii. Are developers familiar with development environment?
ix. What is the reliability of the build?

e) Testing

i. Are we equipped with automated testing?
ii. Is the bug tracking system effective?
iii. What is the test coverage?
iv. What metrics are used to evaluate features?
v. Do we analyze tests?
vi. Is enough time allocated for test cycle?
vii. Are test goals achieved?
viii. Are the resources allocated for testing efficient?
ix. Do we have required system resources?
x. How do defects affect testing life cycle?

f) Maintenance

i. Is the defect severity order defined?

ii. Can the bugs be reproduced?

iii. How complex and critical are the bugs?

iv. What is the scope for testing of a bug?

Products and By Products

a) Documentation

i. Is the documentation adequate?

ii. Is documentation clear and legible?

iii. Is the documentation consistent?

iv. Are documents delivered timely?

v. Do documents cover all the requirements?

vi. Are we producing user/training manuals?

vii. Do we document all the defects?

b) Source Code

i. Is our source code readable and maintainable?

ii. Do we have enough comments within source code?

iii. Is source code documented?

iv. Do we have log bug rate?

c) Future plans

i. Are plans consistent with customer requirements?

ii. Are product changes communicated to customer?

d) Budget

i. Did we over allocate resources?

ii. Do we have enough slack associated within the project?

iii. What is our cost and schedule index?

iv. How do bugs/failures affect budget?

Step III: Identify the sub-goals

The main goal was then divided into sub-goals related to activities or tasks performed during the software life cycle. These sub-goals were grouped as follows:

a. Resources – helps determine the resources (intangible and tangible) that the company needs to set aside.
b. Project Management – helps identify the potential problems and set metrics for overall success of the software.
c. Documents – documenting the process will help in benchmarking the artifacts.
d. Environment – helps identify factors that could adversely affect development.
e. Software Design – a strong design eliminates reworking during the development cycle.
f. Source code/final product – helps controls the changes that are made during version development.
g. Process – a well-defined process will help determine the allocation of the resources and time to ensure maximum productivity.
h. Communication – a better understanding of the data that is being collected is essential for the overall success of the project.
i. Others – all plans should be consistent with customer specifications.

Question Groups

a) Resources
    i. Is hardware compatible?
    ii. Is hardware configurable?
    iii. What is the hardware maintenance cycle requirement?

iv. Do we have competent development team?

v. Do our developers like their jobs?

vi. Are the developers working across projects/clients?

vii. Use of an IDE and its effectiveness?

viii. What is the frequency of Third party tool usage?

ix. Do we reuse code from past projects?

x. Is your development environment up to date?

xi. Do we have required system resources?

b) Project Management

i. Do we have past experience in project domain?

ii. Do we have dedicated business analysts?

iii. Does the team adhere to project schedule?

iv. Do we have control over requirements?

v. How complex are new requirements?

vi. When do we say no to new requests? Do we have concept of requirements freeze?

vii. Are you utilizing past metrics for resource cost analysis?

viii. How do bugs/failures affect budget?

ix. What techniques/models are used in estimation?

x. Can we support our estimates with real data?

xi. Are there any deadlines?

xii. Is enough time allocated to design analysis?

xiii. What is the test coverage?

xiv. Is enough time allocated for test cycle?

xv. Are test goals achieved?

xvi. Are the resources allocated for testing efficient?

xvii. Did we over allocate resources?

xviii. Do we have enough slack associated within the project?

xix. What is our cost and schedule index?

xx. How do defects affect testing life cycle?

c) Documents

I. Do we have a documentation team?

II. Do we have existing metrics from past projects?

III. Are the design documents logical?

IV. Is the defect severity order defined?

V. Is the documentation adequate?

VI. Is documentation clear and legible?

VII. Is the documentation consistent?

VIII. Are documents delivered timely?

IX. Do documents cover all the requirements?

X. Are we producing user/training manuals?

XI. Is source code documented?

XII. Do we document all the defects?

d) Environment

i. Do the developers like their jobs?

ii. Is our staff motivated and challenged?

iii. Is the work environment ideal?

iv. Are the developers equipped with proper tools to perform their jobs?

v. How is the interaction between development and QA team?

vi. Are we equipped with automated testing?

vii. Is the bug tracking system effective?

e) Software Design

1. Is the development, architecture and design team involved in requirements gathering?
2. Is product design communicated within development/engineering team?
3. Is our design comprehensible?
4. Does design satisfy architectural requirements?
5. How is design evaluated?
6. Is code analysis a part of software design?
7. How do defects affect design?

f) Source Code / Final product

I. Are code standards enforced?
II. Do we perform code reviews?
III. Do we perform proof of concept for a feature?
IV. How efficiently do we use source control?
V. Are developer changes tracked?
VI. How complex and critical are the bugs?
VII. Is our source code readable and maintainable?
VIII. What is our failure rate?

g) Process

i. Do we have an established process to collect requirements?
ii. Do we have a process to collect metrics during project development?
iii. Do we have a process to acquire commitments on requirements?
iv. What is the frequency of requirement change?
v. Is there a correlation between number of bugs and changing requirements?
vi. Are the requirements changing close to deadlines?
vii. Is the development and testing environment integrated?

viii. How often are requirement triaged?

ix. Is empirical data being considered for estimation?

x. Do we have an efficient build system?

xi. Do we have test analysts?

xii. Can we reproduce bugs?

h) Communications

i. Do we comprehend our software measurements?

i) Others

I. Are plans consistent with customer requirements?

Sub-goals from Question Group

a) Produce quality documentation

b) Improve design quality to avoid failures, errors and conflicts

c) Predict timely estimates for scheduling release

d) Estimate and reduce defect probability

e) Improve testing process

f) Improve Quality and Reliability of the system

g) Identify defect density and reliability of the overall system.

Step IV: Identify entities and attributes

After finalizing the sub-goals in Section III, the mental models were analyzed to identify the entities and attributes associated with them. As per the current software measurement project requirement, sub-goals related to defect density and reliability of the system were analyzed. The following questions helped identify the entities and attributes related to defect density and reliability of the overall system:

Question 1.

How does a defect affect design?

Entity:

All the code svn (Subversion) check-in for each bug/failure reported/found.

Attributes:

Length of source

Uncommented Length of code

Number of changes to the code

Number of defects logged

Cohesion/Coupling across various classes

Question 2.

How complex and critical are the bugs detected?

Entity

All the defects opened during and after the release of the product.

Attributes:

Length of source

Uncommented length of code

Number of affected releases

Number of long-lasting bugs

Affected functionality changes to the code

Cohesion/Coupling of the affected classes

Question 3.

Do we efficiently use source control?

Entity:

Source control system SVN, JIRA logging and build system ANT-Hill

Attributes:

Number of branches created

Amount of rework on class

Number of comments

Number of affected bugs fix checked into various release

Comment to check-in ratio.

Question 4.

How do defects affect testing life cycle?

Entity:

All the bugs opened during the QA release testing life cycle

Attributes:

Number of defects opened

Number of fixed defects

Defect severity

Number of regression defects

Number of high severity bugs.

Number of functionality changes

Check-in to functionality ratio

Available amount of time to address/reproduce the issue

Question 5.

Is the source documentation adequate?

Entity:

All the bugs opened for the life cycle of the release, including pre-release bugs

Attributes:

Release date

Number of defects opened after and before release

Number of fixed defects

Severity of defects

Step V: Formalize measurement goals

Upon identifying the entities and the attributes, the sub-goals were translated into clear measurement goals. The translation was achieved by defining: objects of interest, purpose, environment and constraints for considered sub-goals. Data transformation (Extraction, Abstraction, Aggregating, Transformation, etc) processes were performed to gain information about the attributes that act as inputs for sub-goals.

Step V a. Data transformation

Many software defect-tracking systems do not provide software metrics or extractable defect information and are not easily manageable. In order to evaluate, calculate or identify information that will help foresee the main objective of this measurement process, data transformation is required.

JIRA is the defect-tracking system used to store all the defects and project-related information in MYSQL database tables. These database tables have a defined proprietary structure. All the data related to one particular defect or feature is distributed across multiple database tables defined within the JIRA system.

Some of the main database tables in JIRA and their useful attributes are shown below

Database Table name: JIRAISSUE, provides

Example data:

ID	pkey	PROJECT	issuetype	PRIORITY	issuestatus	CREATED	UPDATED
15022	ONQ-1633	10020	Bug	Trivial	Open	1/29/09 13:50	4/13/09 16:31
15026	ONQ-1637	10020	Bug	Trivial	Open	1/30/09 11:02	5/28/09 10:20

Each row represents single defect opened.

Column Info:

ID: Unique identification for the row

pkey: Unique key the defect

PROJECT: key to join PROJECTINFO database table

PRIORITY: severity of the defect

Issuestatus: current status of the defect

CREATED: date and time the defect is created

UPDATED: data and time the issue was last updated

Database Table name: NODEASSOCIATION, provides defect fixed release number

Example data:

ID	TIMEORIGINAL ESTIMATE	TIMEESTIMATE	TIMESPENT	WORKFLOW ID	Project ID	Release
13390				13390		
13394				13394	10020	3.0.0
13395				13395	10020	3.0.0

Each row represents association with the software release affected for a defect i.e., one row per every release affected by a single defect

Column Info:

ID: Unique identification fo the row

TIMEORIGINALESTIMATE: estimate provided to address the defect

TIMEESTIMATE: amount of time taken to address the issue

WORKFLOWID: foreign key for JIRAISSUE table

ProjectID: key to join PROJECTINFO database table

Release: column join with RELEASEINFO table.

The tables JIRAISSUE and NODEASSOCIATION are joined over the columns WorkflowID and ProjectID to draw relation between the issue and project affected attribute.

Likewise various other data extraction, sampling and transformations are performed over multiple database tables to obtain the required data.

Step V b: Formalizing sub-goal invariants

Sub-goal 1: Source Code / Final code

Object of Interest

a) Defect density of the system

Purpose:

Define and calculate defect metrics to estimate failure intensity

Perspective:

Study the failure/bug data to predict defect metrics to understand defect density across various releases

Environment:

The defect intrusion rate estimated using JIRA bug tracking information across various products and releases helps in making informed product release relative decisions.

Constraint:

Entire defect data gathered has defect addressed for various releases.

Sub-goal 2: Software design

Object of Interest:

Improve design quality by reducing defect density

Purpose:

Evaluate design quality of each release reducing the probability of defect injections thereby reducing rework

Environment:

Anthill and SVN used in combination keeps track of code check-in for each defect. Developers and test teams, who verify the information, need to understand the quality of the release and the overall integrity of the system.

Constraint:

Check-in performed for various releases that is identified as defects.

Sub-goal 3: Software schedule

Object of Interest:

Estimate product release schedule effectively/precisely.

Purpose of Interest:

Elicit defect identification rate to schedule resources as per need. Thereby reducing the need to accommodate time to address the issues and testing life cycle.

Constraint:

Most of the defects/failures/bugs reported affect release life cycle of various systems. Need to segregate testing life cycle for various releases and time taken to address the issue on releases basis.

Step VI a: Identify quantifiable questions and indicators

Quantifiable Questions:

a) What is our defect density for each release?
b) What is the failure intensity?
c) Are the trends indicating failure/defect intensity? How do we interpret it?
d) What are the complexities across release?
e) What is the timeline of defects for release?
f) What is the cumulative growth pattern of the failures?

Step VI b: Formal data representation for defect density analysis

The number of known defects is the count of total defects identified against a particular software entity during the period of the release. Defect density is calculated as follows:

$$\text{Defect Density} = \frac{\text{Number of Defects}}{\text{Size}}$$

Test1: Change in Requirement – Defect Density

The software development life cycle starts with the gathering of requirements. Requirements change as per the need of the customers. Initial release of the software developed is based on the set of requirements aggregated from various customers. When the software is released, customers evaluate it as a part of UAT (User Acceptance Test). During this process all the defects, failures and functional shortcomings of the software are logged into the defect-tracking system (JIRA). Functional shortcomings

logged in JIRA are categorized as “Improvement” or “New Feature” and are considered for post-release defect density analysis.

Fig I below, compares defect density for ‘*no change in software requirements’* verses ‘*change in software requirements’*.

This figure indicates that the defect density of the software increases as the number of changes in software requirements increases linearly. This indicates the importance and need for solidifying the requirements-gathering process prior to design and implementation phases in the software development life cycle.

OnQ version 3.2.0 included many new features and functionalities to be implemented based on the requirements gathered more than a year to its release. As the features were interlinked, the requirements changed as the software evolved. This change in requirement was not considered during the implementation of the software. Due to this reason, the defect density is predominant in this release.

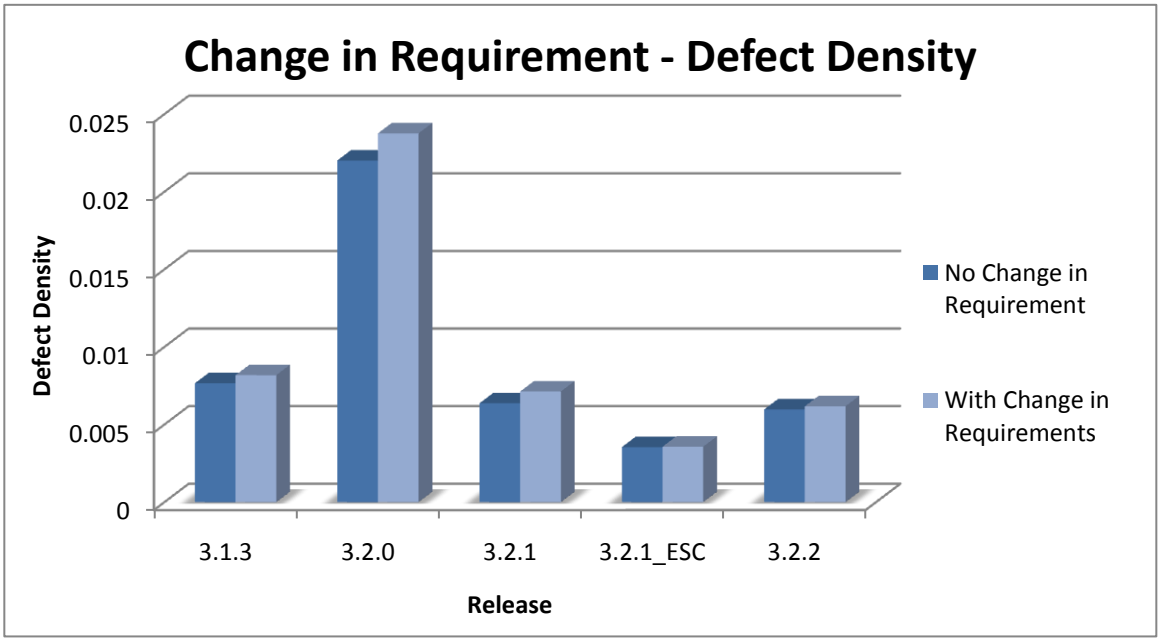

Fig II: Change in Requirement – Defect Density

Test2: Documented Code – Defect Density

The second process that is important in the software development life cycle is design and feature development based on the requirements gathered. Lines of source code being written needs exquisite documentation. This documentation helps developers to understand feature functionality, contributing variables, defined objects etc, for the entire product development life cycle. Also, the product software source code being written should follow certain documentation and coding principles.

During the development and testing processes, the product features are tested using Unit, Integration and Manual testing procedures. All the defects discovered within each release are opened and separately categorized as a BUG in the defect-tracking system (JIRA).

Fig II below uses the entire bug information found during the development and testing process in pre-release versions of the software. As the software evolves over time, the amount of documentation for each module/feature increases. From the figure, we notice a linear change in defect density as the product documentation improved over every release of OnQ. This indicates the importance of documentation during the feature design process.

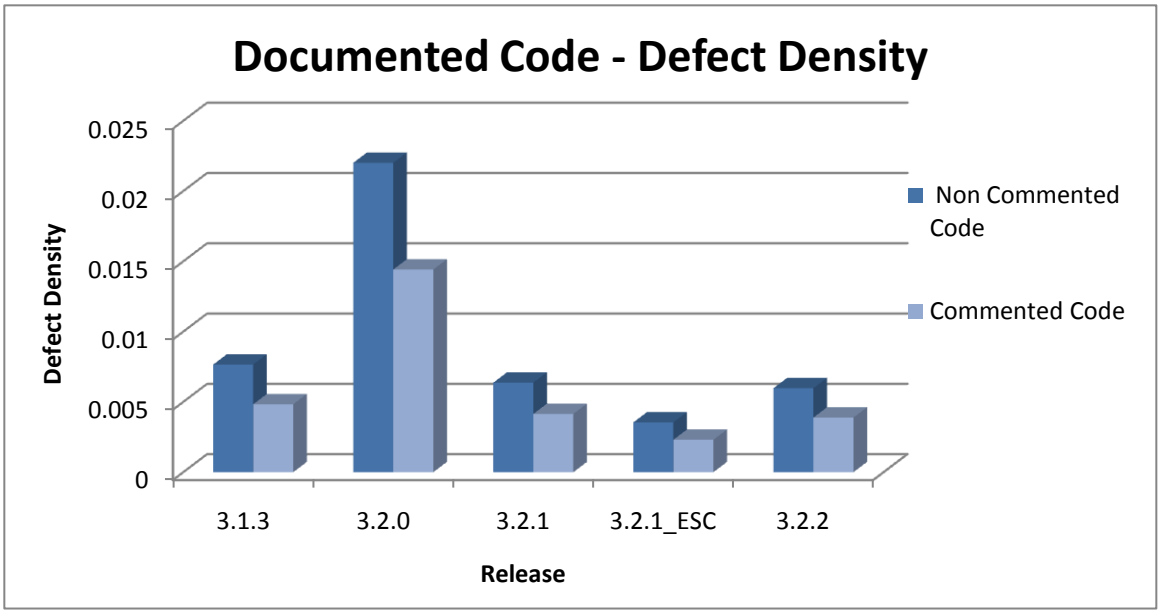

Fig III: Documented Code – Defect Density

Test3: Release-to-Release, source Defect Density

As developers finalize the implementation of a particular feature in the software development life cycle, the number of lines of code increases per the requirements and the documentation. As the product reaches a minimal functional stage as specified in the PRD (Product Requirement Document), the software is subjected to various Unit, Integration and Manual testing procedures. All the defects/failures found within the software (both pre- and post-release) are then logged into the defect-tracking system (JIRA) categorized as defects each with its associated severity indicator.

Before and after the release, only failures of the software are logged into the defect-tracking software (JIRA) as bugs. NLOC (usable lines of code) is calculated using only the lines of directives that are actually functional within each release. Clover, code

coverage and SLOCCount tools are used to determine the total source lines of code for each release.

Below table shows the lines of code for each release of OnQ:

Metadata:

Version: Version number of OnQ product

LOC: number of lines of code, including comments and other directives

NLOC: number of lines of executable code

Version	LOC	NLOC
3.1.3	39518	24938
3.2.0	56725	37163
3.2.1	60807	39557
3.2.1_ESC	60661	39500
3.2.2	60706	39507

Fig III illustrates a gradual decrease in defect density for lines of directives with each release, thus helping understand the quality and integrity of the system with increasing lines of code.

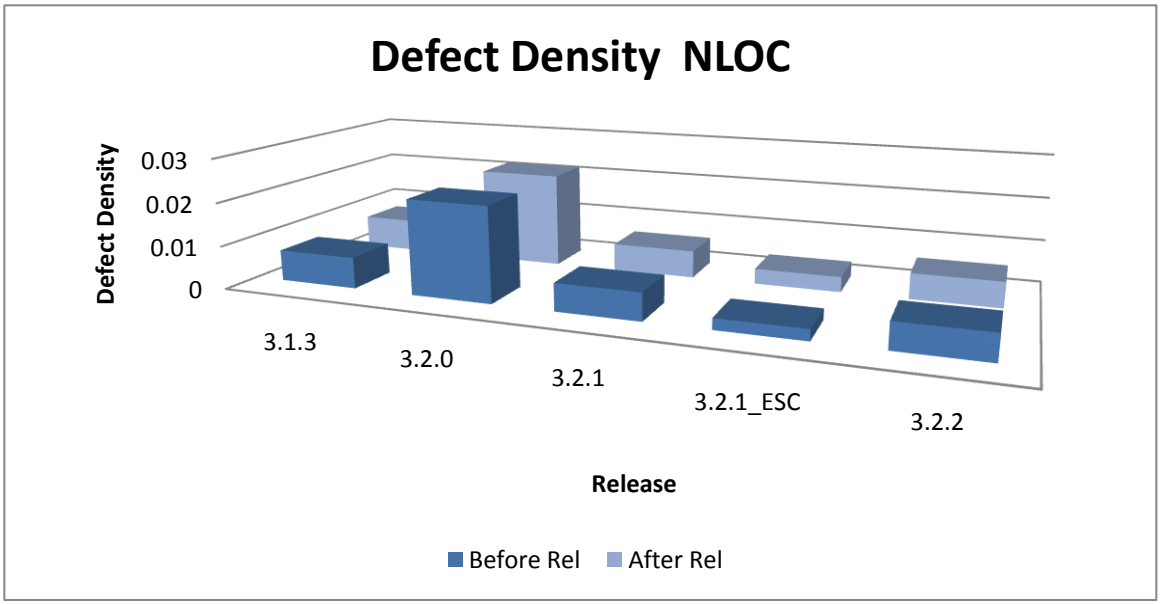

Fig IV: Release-to-Release NLOC Defect Density

Below Fig IV, illustrates the rate of defects per line of code including comments and other non-evaluated directives written within the software source code. Again, the numbers for lines of code were obtained by using Clover and other software source count tools. The below graph helps identify release candidates for additional inspection or testing or possible reengineering.

As we notice from the below Fig IV, the defects across release seem to be trending down for every release cycle when no source documentation was involved. Also the numbers of defects after release of the product seems to be considerably lower for each release, helping to see the improving quality of the system. Hence explaining the importance of the documentation and comments within the overall system.

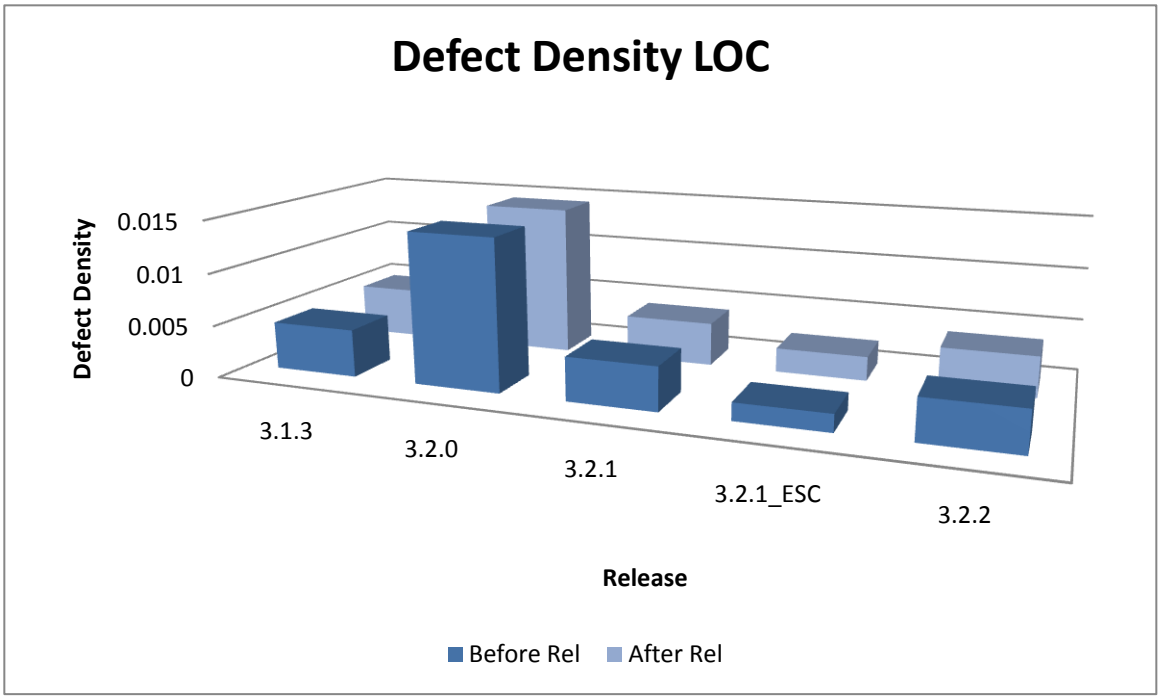

Fig V: Release-to-Release LOC Defect Density

Test4: Product timeline – defect density

Every software product or tool needs to evolve with time. As the software development life cycle progresses, products are finalized and then released once the software meets the PRD (Product Release document) criteria. As the product is worked upon to incorporate new features or improvements, information about probable growth in bugs, rate of failure and its failure intensity timeline will help project and QA managers manage future resource availability and software release schedule.

The following scatter plot (Fig V) shows the time interval between bugs in hours.

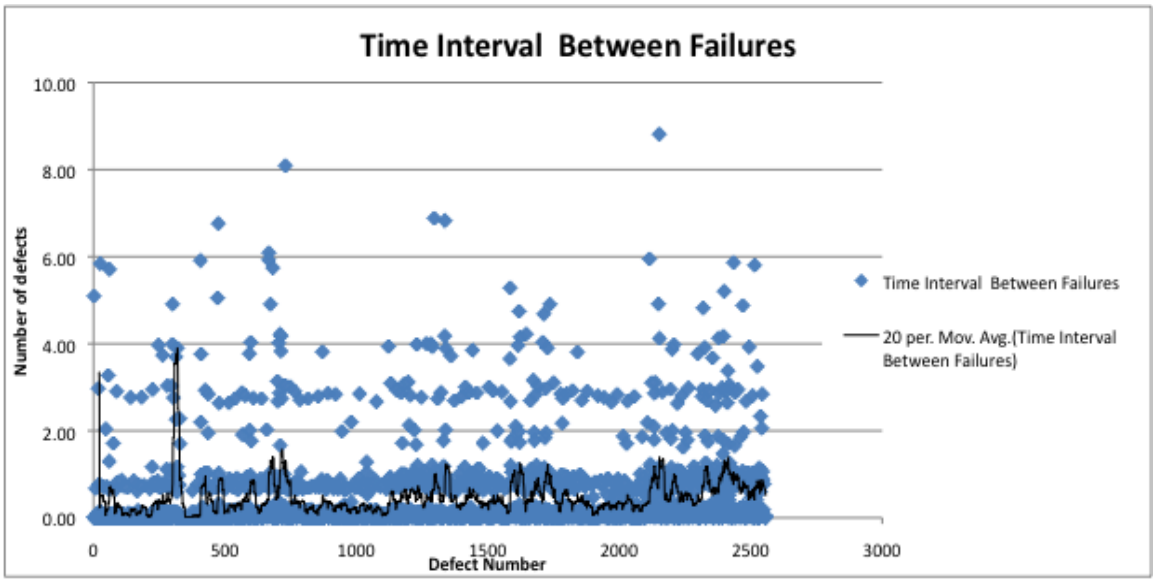

Fig VI: Time Interval Between Failures

The above graph smooth's the defect data to form a defect trend following indicator filtering out the noise in defect-tracking data (JIRA). The rise in trend line indicates the increase in defects opened.

Comparison of subsequent releases of a product to track the impact of defect density provides valuable insights into quality improvement activities. Normalizing by averaging the size of the software allowed releases of varying sizes to be compared in Fig VI. This chart can help executives and managers to estimate future product release timelines for scheduling

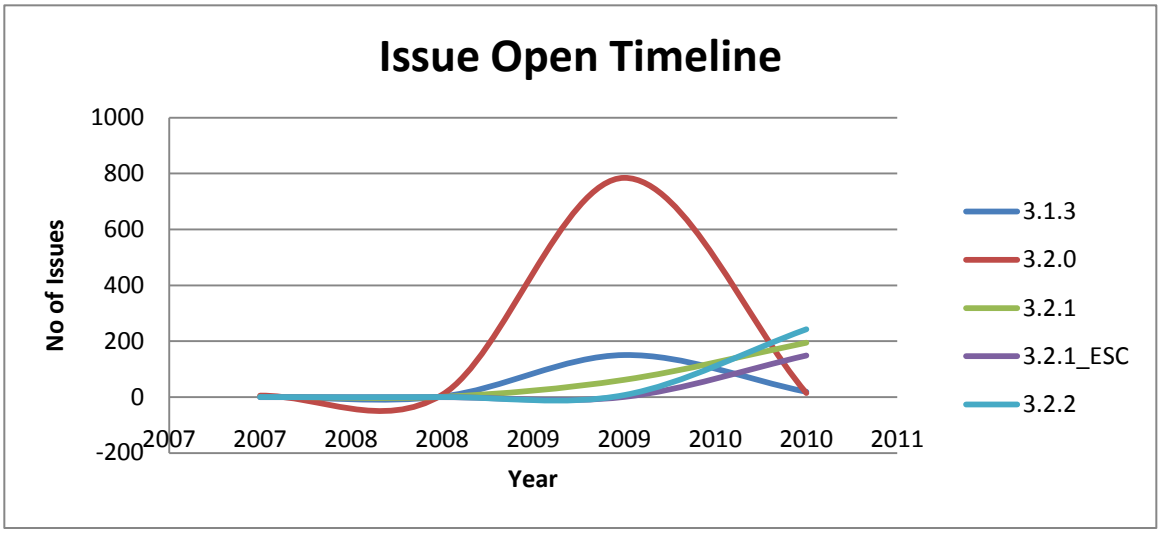

Fig VII: Product track yearly

Step VII. Identify data elements

The following table shows the data elements used for the indicators defined in Section VI b. 'X' mark denotes that the data element was used during the indicators creation.

Data Element	Indicators					
	Change in Requirement	Documented Code	LOC	NLOC	Time Interval Between Failures	Issue Timeline
Feature Area			X			
Data Opened	X	X	X	X	X	X
Interval since last bug	X				X	X
Date Resolved	X	X	X	X	X	
Data bug completed			X	X		
Priority			X	X		
Source lines of code	X	X		X		
Number of classes						X
Sources lines of code uncommented		X	X			
Release dates			X	X		X
Number of source code changes						X
Number defects found for area		X				X

Step VIII: Define measures

A software measure is undefined until its need and its usage is understood by the organization. The below section lists the data elements and their respective scale considered during creation of the indicators in section VI b.

Time:

Time is in hours

Date Stamp:

Date the issue was opened

Date the issue was resolved

Date the issue was completed

Date the issue was closed

Software release dates

Cumulative dates between failures (Includes holidays and vacation)

Lines of code:

Declarations

Compiler directives

Number of Comments

Number of references

Number of classes and objects

Number of generated code

Reused number of lines

Code release to be compatible with no errors in build cycle

Use only the changed code for metrics

Function and feature points to evaluate number of tests to address

Defects:

Number of defects per area/functionality

Change in time between issues opened

Defect density across timeline

Change Requests

Feature enhancements that come in from the customer and require release or test life cycle adjustments to accommodate addressing and testing of new bug/failure.

Quality Analysis

Defects uncovered during testing

Duplicate issues

Related issues

Automated test failures

Count inputs, outputs and interfaces

Test density for feature

Step IX: Identify the actions needed to implement measures

For the measurements process to continue, translation of all the above definitions and processes need to evolve into action plans. The below table indentifies the ease of obtaining the information and the availability of the data elements used in Section VI b.

Scale used for the below table:

+ - indicates readily available

++ - indicates the need for data format change

0 - indicates the need for minor effort for data extraction

Data Element	Availability	Source
Feature Area	+	Code, SVN
Date Opened	++	Defect Tracking System(JIRA)
Date Resolved	++	Defect Tracking System(JIRA)
Date Completed	++	Defect Tracking System(JIRA)
Date Closed	++	Defect Tracking System(JIRA)
Priority	+	Defect Tracking System(JIRA)
Defect Density	+	CASRE, Excel
Source lines of code	0	SVN, Code, Clover
List of Projects	0	JIRA, Test Link
Total number of bugs	+	Defect Tracking System(JIRA)
Release Date	+	SVN, JIRA, ANT-Hill
Automated Test issue	+	ANT-Hill, Automated Build system

Below is an overview of how the data element associated with source could be obtained.

a) Feature Area: Areas of the software – this can be known by the area attribute in SVN check-in or defect-tracking system (JIRA) code check

b) Date Opened: This is readily available from the “date Opened” attribute of the defect-tracking data (JIRA)

c) Date Resolved: This is readily available from the “date Resolved” attribute of the defect-tracking data (JIRA)

d) Date Completed: This is readily available from the “date Completed” attribute of the defect-tracking data (JIRA)

e) Date Closed: This is readily available from the “date Closed” attribute of the defect-tracking data (JIRA)

f) Priority: Available from defect-tracking (JIRA) database table.

g) Defect Density: Calculated using Microsoft excel macros

h) Source Lines of code: Calculated using source analysis tool on source code

i) List of projects: Available from defect-tracking (JIRA) and build system (ANT-Hill)

j) Total Number of bugs: Available from defect-tracking system (JIRA).

k) Release date: Available from joining defect-tracking (JIRA) information to build system (ANT-Hill).

l) Automated Test issues: Available from automated build systems (ANT-Hill)

Step X: Measurement implementation plan

Successful completion of the software measurement implementation is dependent on effective planning. Below project plan was developed and followed to define the software measures that satisfied the identified business goal.

a. Objective:

This project intends to calculate defect density and defect metrics and in process help identify probable improvements in the software development life cycle. It will also help executives and developers to focus on any negative trends that may raise red flags in time to implement mitigating measures. Some of the crucial measures that needed for analysis of quality of the system include

- Defect density
- Defect Slippage rate
- Failure Intensity Rate
- Reliability

These metrics provide

- better release date probability calculation
- mitigate complexity risk
- reduce resource movements/allocations
- build confidence within product
- assess the stability of the release etc

b. Background Description:

The predicted quality, reliability and release schedule of the system were constantly incorrect, due to lack of appropriate measurement models. Our company had no/low understanding of how stable was our product. In these competitive and cash strapped oriented market a reliable and stable system is a must for any software service oriented company. This leads to implementation

of defect related software measurements metrics to evaluate defect density within the system. This metrics calculated will ideally act as an input to management to realistically assess the robustness and integrity of software. It also helps them make accurate recommendations on the time to market date of the release/product.

c. Goals

1. Business Goals:
    i) Company standing to provide reliable collection system.
    ii) Improve product release date estimation with a known defect density measure
    iii) Effective resource allocation
    iv) Decrease cost be predicting defect density measurement for every release.

2. Measurement Goals:
    i) Improve accuracy, speed and effectiveness of product release schedule
    ii) Increase confidence in product
    iii) Sustain sudden increase in the number of failures by estimating failure intensity and density

3. Goals of this project:

    The objective will be realized when the defect density of the software is reduced and is below the threshold level set by the business executives.

d. Scope

Real-time defect data that was generated during the development period of over 2 years prior to the first release of the product. Each defect had an exquisite set of attributes that included date of issue opened, closed, completed and resolve, severity of the bug, changes associated, software area affected, priority etc. Access to source code also helped in gathering the actual code relative metrics such as source lines of code both commented/uncommented, area and functionality affected etc. Issue being addressed in multiple releases needed considerable amount of work for associating the issue within different releases.

These metrics will enable us to compare historic releases and software integrity growth for every release. It also provides us valuable information to help predict release cycle effectively.

Various beneficiaries of the successful implementation of measurements include

- Product Managers
- Project Managers
- Developers
- QA

Defect metrics calculation provides direct relationship of defects to the integrity of the product thereby affecting the software improvement process efforts. Also helping various managers and QA to predict probable release schedule.

e. Implementation

1. Activities, Products and Tasks

a) Establish realistic timelines based on regression cycle and failure intensity rates

b) Tie-up the bug tracking system and source control system to generate defect density for every release.
c) Identify negative trends in defect density

f. Measurement and Monitoring

Using code metrics, particularly defect density we have first hand information on integrity growth of the product.

g. Assumptions

Few major assumptions include

- One defect per release
- One fix per release
- Issues are entered when found with no delay
- All the provided data is appropriate

h. Sustained Operations

The measurement system will be evaluated consistently for success based on information obtained about system integrity.

Conclusion:

The quality and readiness of the application, and its approximate release schedule, has always been a persistent problem within ALI. Inc Solutions. Assuming that we can measure the quality of the system by analyzing the number of defects opened, we examined several related software lifecycle issues. These are: the frequency of changes to the requirements, lack of test coverage and deficiency in source code documentation. These issues created additional overhead to the cost, reduced resource availability and incorrectly predicted the release schedule, thus lowering the integrity and reliability of the overall system. Many of these attributes could be comprehended by designing a simple scorecard program, which directly relates to business goals and objectives. Measuring a software application matters only if we can show how the measures are related to one's organizational goals and objectives; and how measuring them will improve the organization or its processes.

The GQM approach has been shown to help establish effective measurement programs by coupling business objectives to metrics through defined the models that focus on the achievement of the business goal [12]. By following the steps defined by the GQM model process, the organization's main objective was abstracted to multiple sub-goals in Section III. Based on the questions that needed answers by various departments described in Section II, each sub-goal was then associated to quantitatively measurable attributes and entities in Section IV. The data available from the defect-tracking system (JIRA) database tables were then extracted and transformed into the required formats for analysis as mentioned in Section V.

The classic software development life cycle model is generally divided into five main categories with their transitions shown below.

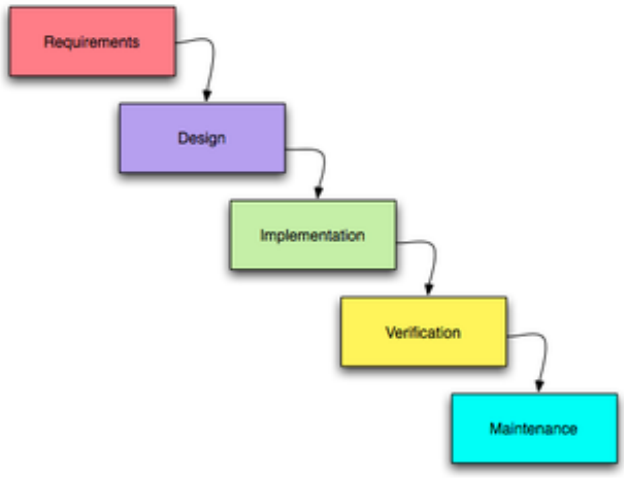

Fig VIII: Classic Software Life Cycle Model

Below results can be defined based on the indicators described in Section VI b

Result 1: One of the most important aspects of the classic software development process is having well defined requirements describing the capabilities that the software must implement. Elaborating the requirements in order to determine the sub-goals the software must provide requires in-depth analysis. Changes in requirements, potentially lead to changes in the design and code of the application. In ou situation, as the number of customers increased, new features and enhancement requests increased, thus requiring changes in application design. Upon analyzing the defect data for 'change in requirement' versus 'no change in requirement' we noticed an increase of **6.37%** in defect density. This predominantly speaks to the need for solidifying the requirements and design for the application prior to launching its implementation process. The software must not be designed just to accomplish the desired function, but it must do so in a manner that best meets the needs of the business. With fewer changes to the software requirements, the amount of rework in design and functionality is also reduced.

Result 2: The design and implementation phases are considered equally important for the software development life cycle. The design and the source code documentation are critical for efficient software development. Insufficient documentation could lead to unjustified assumptions about the software source code. It is observed that without applying design and documentation principles, the application development process ended up with common software problems (i.e. scope creep, poor functionality, difficulty to reuse, impossible to maintain code). With no ground rules laid for documenting the software design and by not implementing appropriate documentation procedures during coding (implementation) defect density increased **33.45%.** Thus emphasizing the importance of attention towards documentation during the design and implementation phases of the product life cycle.

Result 3: Verification/validation of an application is considered to be a critical phase of the software development life cycle. The importance of proper verification processes is seen in the number of defects opened during the pre and post-release of the application. Considering "no change" in the application requirement, a 6.9% increase in defect density was noticed between the pre and post release of the application. This identified a gap in test scenarios covered during application readiness testing. Hence a defined validation process identifying all the invariants to test for each module in the application would help lower the change in defect density level.

Some of the defects, enhancements or new feature requests were opened due to any of these reasons: incorrect requirements, inappropriate design and documentation causing a defect, and missing testing coverage. Also, it has been observed that a deficiency in multiple phases of software life cycle has resulted in a single/common defect. So, on normalizing the defect density values by averaging for requirements, design, implementation and verification processes, the results obtained are shown below in Fig VIII.

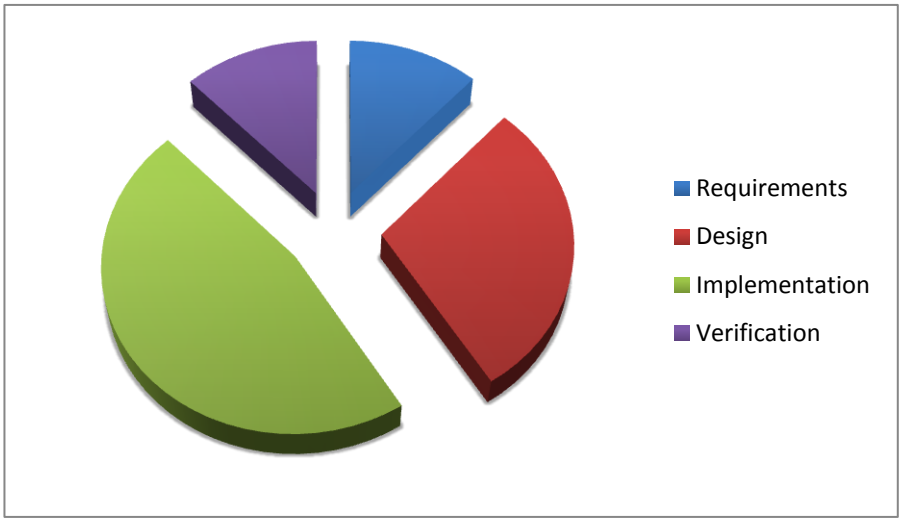

Fig IX: Defect Density – Software Life Cycle relation

The Implementation phase is the most significantly affected phase of the software life cycle by the defect density, thus explaining the need for following documentation principles for the defect density reduction. The defect density, explaining the need for documenting design, solidifying requirements and verifying test mapping respectively, also affects the design, requirements and verification/validation phases.

The results above have direct implications for the software lifecycle. These are described below.

- The software requirements process acts as a basis of agreement between the customer's objective and the software product delivered. Changes in requirements alter the software environment and the relationships between its developed features. With the software development process being so complex, it is risky to embark on design before knowing what to build. So it is important to identify all the software requirements and their needs in detail prior to the launch of the design phase. This is the assumption of the classic

lifecycle model. If this assumption proves to be false in the future, other lifecycle models must be considered.

- The software design phase specifies the required and optional functionality to achieve the customer goal; it also helps identify the completeness of the functionality. Without clear specifications regarding the goal, implementation of the software is unguided. Thus it is difficult to know when the program is complete and impossible to accurately measure progress. Also, design of the software defines the system elements and how they interact. It is difficult to change/manage them without proper design and its documentation. Furthermore, defining the software design to be solidified, and documenting the source code accurately helps to bridge the gap between requirement and implementation phases.
- Documentation of the source code during the implementation phase reduces the probability of making dangerous assumptions while scrutinizing the design implementation. Regardless of the intent of the software developer, all source code is eventually reused, either directly or just through the basic need to understand it. Software source documentation is an irreplaceable necessity, as well as an important discipline that increases development efficiency and quality. It helps ease adaptability and understanding of the software-reducing rework. Good coding standards are a must.
- The verification phase of the software development life cycle helps identify the quality of the software by testing the agreement of the customer's goal with the requirements, design and features developed. This testing of product fitness of purpose is needed to prove or disapprove the correctness of intended functionalities within the system prior to release of the software. Thus verifying the entire test mapping to functionality and possible invariants during test case development, reduces the amount of rework during maintenance phase of the software life cycle.

Finally, due to lack of use of appropriate defect analysis models, our company had no/low understanding of how stable and reliable the application was/was not. By using the GQM approach, we were able to explain some of the weaknesses and shortcomings in our application development processes.

Future Work:

Measurement program: Multiple sub-goals were identified as a part of Section III described above - based on questions identified by various organization personnel. Concentrating on a few of the identified sub-goals, the defect density metric was selected for analysis as the primary variable in the identification of possible changes in the software development processes.

This process of measurement is assumed to be an ongoing process. It involves adding new tools, plug-ins and features to existing software to collect additional relevant metrics.

Some of the metrics under consideration are

- Cyclomatic complexity – to measure complexities of the source code
- Halstead Volume – to measure difficulty levels of the source code
- Coupling factors – to verify level of dependency in source code
- Cohesion factors – to verify level of strong relation in functionality
- System reliability - to verify the ability of the system to perform required functions
- Automated and Manual test coverage – to identify missing test scenarios
- Mean time to recover – to uphold maintenance contracts, this metric will help identify the amount of time it will take to recover/fix the defect
- Mean time to failure – elapsed time between inherent failures for resource scheduling tasks

Process Futures: In light of the issues uncovered in this research, it may be necessary to rethink our lifecycle model choice. Perhaps agile and/or lean methods will need to be investigated.

References

1. Klein, Hartmut, 2007 “Basic Project Planning”
2. Wysocki, Robert K. March 27, 2006 – “Effective software project management”
3. Capers Jones , Third Edition, 2008– “Applied Software Measurements”
4. GQM Handbook - https://goldpractice.thedacs.com/practices/gqm/
5. ALI Solution – Proprietary product scope and design documents of the product.

   Software Measurement and Formal Methods: A Case Study – Lieonel C Briand
6. Requirements Engineering Course work – Dr. Kathleen Barber
7. Project Management Course work - Dr Robert McCann
8. 2002, Basili,V.R., and D.M.Weiss,” A methodology for collecting valid software engineering Data”,IEEE TOSE,vol.
9. SE-10,pp.728-738, 1984. 03. Basili,V.R, et al, Goal Question Metric Approach, John Wiley&Sons,1994. 04. Wolfhart Goethert, Deriving Enterprise-Based Measures using the balanced Scorecard and Goal-driven measurement Techniques,
10. vol.37,No.4,1998. 06. Markus Nick et al,Facilitating the practical Evaluation of Organizational Memories using the Goal-Question-
11. Metric Technique, KAW’99,1999.
12. Wiki - http://en.wikipedia.org/wiki/GQM